Module 2: The basics of Terraform Language (HCL)

Terraform level 1 - Day 1 - **Module 2**

Agenda

From Concepts to Code

In Module 1, you learned:

- What IaC is
- Why Terraform is valuable

Now we move from **concepts** → **real Terraform code**.

Key idea:

👉 All Terraform automation happens inside **.tf files** written in **HCL**.

What is HCL?

HCL = **H**ashiCorp **C**onfiguration **L**anguage

- Human-readable syntax for writing infrastructure configurations
- Think of it like a specialized language for describing what your infrastructure should look like
- Similar to JSON or YAML, but designed specifically for infrastructure

Why HCL matters for banking:

- Clear, auditable configurations for compliance
- Easy for teams to review and approve changes
- Self-documenting infrastructure code

Key HCL Building Blocks

Every Terraform configuration is built from two main components:

1. **Arguments** (Settings) - Individual configuration values
2. **Blocks** (Sections) - Containers that group related settings together

Let's explore each one…

Arguments (Configuration Settings)

What are arguments?

- Arguments are the individual settings that configure your infrastructure
- Think of them like form fields you fill out: *setting name* = *value*

Simple example:

```
region = "us-east-1"
```

- `region` is the **argument name** (what you're configuring)
- `"us-east-1"` is the **value** (what you're setting it to)

Arguments - More Examples

From real banking scenarios:

```
instance_type = "t2.micro"      # Small instance for dev/test
```

```
instance_type = "m5.xlarge"     # Larger instance for production apps
```

```
bucket_name = "compliance-audit-logs-2024"
```

Key point: Arguments let you specify *exactly* how each piece of infrastructure should be configured.

Arguments - Value Types

How does Terraform know what type of value to expect? The

context determines the type:

```
resource "aws_instance" "trading-app" {
  instance_type = "t2.micro"     # This must be a string
  monitoring = true              # This must be true/false
  count = 2                      # This must be a number
}
```

- The `aws_instance` resource defines what each argument accepts
- Terraform validates your values automatically
- **Benefit for banking:** Prevents configuration errors before deployment

Blocks (Configuration Sections)

What are blocks?

- Blocks are containers that group related settings together
- Think of them like sections in a form, each with its own purpose
- Most Terraform features are implemented as blocks

Structure of a block:

```
block_type "label1" "label2" {
  # Arguments go inside here
}
```

Blocks - The `resource` Block

The most common block type is `resource` - this defines infrastructure to create:

```
resource "aws_instance" "compliance-server" {
  instance_type = "t2.micro"
  ami = "ami-12345"
}
```

Breaking it down:

- `resource` = **block type** (what kind of thing this is)
- `"aws_instance"` = **first label** (specific resource type from AWS provider)
- `"compliance-server"` = **second label** (your chosen name for this resource)
- `{ ... }` = **block body** (contains all the arguments/settings)

Important: Each resource must have a unique name within a the same resource type. For example:

```
resource "aws_instance" "web" {
  # ...
}

resource "aws_security_group" "web" {
  # ...
}

# This one is invalid since we already have
# an aws_instance with a web label
resource "aws_instance" "web" {
  # ...
}
```

Blocks - Nested Blocks

Some blocks can contain other blocks inside them:

```
resource "aws_instance" "banking-app" {
  instance_type = "t2.micro"

  # Nested block for network configuration
  network_interface {
    device_index = 0
    subnet_id = "subnet-abc123"
  }
}
```

Why nested blocks?

- Organize complex configurations into logical sections
- Make code easier to read and maintain
- Group related settings together (like all network settings)

Types of Values in Arguments

Arguments can accept different types of values:

1. **Simple values** (Literal expressions):

```
instance_type = "t2.micro"                # String
port = 443                                # Number
enable_monitoring = true                  # Boolean (true/false)
```

2. **Collections** (Groups of values):

```
availability_zones = ["us-east-1a", "us-east-1b"]    # List
tags = {                                              # Map
  Environment = "production"
  Department  = "trading"
  CostCenter  = "IT-12345"
}
```

Advanced Value Types

3. References (Using values from other resources):

```
subnet_id = aws_subnet.main.id     # Reference another resource
```

4. Dynamic expressions (Calculated values):

```
instance_count = var.environment == "prod" ? 3 : 1   # Conditional
combined_name = "${var.app_name}-${var.environment}" # String interpolation
```

For beginners: Start with simple values. You'll learn references and expressions in later modules.

Understanding Providers

What are providers?

- Providers are plugins that enable Terraform to interact with cloud platforms and services
- Think of them as "translators" between Terraform and AWS/Azure/GCP/etc.
- Each provider adds specific resource types you can create

Why providers matter:

- Without a provider, Terraform doesn't know how to talk to AWS
- Different providers for different platforms (AWS, Azure, databases, monitoring tools)
- Banking organizations often use multiple providers (cloud + SaaS services)

Declaring a Provider

You must tell Terraform which providers your configuration needs:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 6.20.0"
    }
  }
  required_version = "~> 1.13.5"
}
```

What this does:

- Downloads the AWS provider plugin
- Locks to version 6.20.x (important for stability)
- Makes AWS resource types available (like `aws_instance`, `aws_s3_bucket`)

What Providers Give You

Once you declare a provider, you get access to:

1. **Resource types** - Infrastructure you can create:

```
resource "aws_instance" "app-server" { ... }
resource "aws_s3_bucket" "audit-logs" { ... }
resource "aws_rds_instance" "customer-db" { ... }
```

2. **Data sources** - Query existing infrastructure:

```
data "aws_ami" "latest" { ... }
data "aws_vpc" "existing-network" { ... }
```

Banking use case: Use data sources to reference existing compliance-approved networks or AMIs.

Data Sources - Querying Existing Infrastructure

What are data sources?

- Data sources let you query and reference existing infrastructure
- Use them to look up information you need but didn't create with Terraform
- Think of them like "read-only" access to existing resources

Why use data sources?

- Avoid hardcoding AMI IDs that change with each patch
- Reference infrastructure managed by other teams
- Ensure you're using approved/compliant resources

Data Sources - Real Example

Scenario: Instead of hardcoding an AMI ID, query for the latest patched version of a specific LTS release:

```
# Query for the latest patched Amazon Linux 2023 LTS AMI
data "aws_ami" "al2023-lts" {
  most_recent = true
  owners      = ["amazon"]

  filter {
    name   = "name"
    values = ["al2023-ami-2023.*-x86_64"]  # Amazon Linux 2023 LTS
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}
```

Data Sources - Using the Results

Now reference the data source in your resource:

```
resource "aws_instance" "fresh-instance" {
  instance_type = "t3.micro"
  ami = data.aws_ami.al2023-lts.id    # Reference the data source!
}
```

Benefits:

- Automatically gets the latest patched version of the LTS release
- No need to update code when security patches are released
- Reduces hardcoded values in your configuration

Banking use case: Reference centrally-managed VPCs, security groups, or encryption keys maintained by your security team.

Terraform Registry

What is the Terraform Registry?

- A central repository for discovering providers and reusable modules
- Like an "app store" for Terraform components
- Available at: https://registry.terraform.io

What you'll find there:

- Official providers from HashiCorp (AWS, Azure, GCP)
- Third-party providers (databases, monitoring tools, SaaS platforms)
- Pre-built modules for common infrastructure patterns

Why the Registry Matters for Banking

Public Registry:

- Browse and use thousands of providers
- Find modules for common patterns (VPC setup, security groups, etc.)
- All directly integrated with Terraform CLI

Private Registry (for your organization):

- Share approved, compliant modules internally
- Maintain security and governance standards
- Keep proprietary configurations private

Example: Your security team could publish a "compliant-vpc" module that everyone uses, ensuring consistent network

Knowledge Check 1

Which HCL component is used to define infrastructure you want to create?

A) Data source

B) Provider

C) Resource block

D) Argument

Knowledge Check 2

Why would you use a data source instead of hardcoding an AMI ID?

A) Data sources are faster

B) To automatically get the latest patched version

C) Data sources are required by Terraform

D) Hardcoding is not allowed

Knowledge Check 3

- What is the correct syntax for a resource block?

```
A) resource = aws_instance "my-server" { }
B) resource "aws_instance" { "my-server" }
C) resource "aws_instance" "my-server" { }
D) aws_instance "my-server" { }
```